HOMEWORK 5

Description of the code:

Features are extracted for each of the audio files (English, Hindi, and Mandarin). We have used librosa to extract the features.

Steps for calculating features:

1. A 2D array is computed with shape of (M,64). For 10 second audio file, it will be 1000. 1000 is our training sequence length.
2. The audio files are sampled at 16000 kHz using Mel filters.
3. A 25 msec frame is chosen with a hop length of 10 msec
4. For each audio file we will chose first 1000 features.
5. Create the labels for each class: English: 0, Hindi:1, Mandarin:2

```
l_eng=np.zeros([164,1])
l_hin=np.ones([41,1])
l_man=np.ones([110,1])*2
label_final=np.vstack([l_eng, l_hin, l_man])
```

```
[ ] label_f= np.reshape(label_final,[315,1])
    print(label_f.shape)
```

```
(315, 1)
```

```
[ ] Train_data.shape
```

```
(315, 1000, 64)
```

6. All the features are reshaped into appropriate dimensions from each language and labels are stacked and a data loader is created. Samples are shuffled and split into train and validation set along with their labels.

```python
top_db=30
train_file = '/content/drive/MyDrive/EE599_HW6/train/'
for subdir, dirs, files in os.walk(train_file):
    for file in files:
      #print(subdir)
        print(file)
        y, sr = librosa.load(subdir + "/" + file ,sr=16000)
        # print(y)
        # intervals = librosa.effects.split(y, top_db=top_db)
        # y_new = np.zeros((1))
        # for interval in intervals:
        #     y_new = np.concatenate((y_new, y[interval[0]: interval[1]]))
        # print(y_new)
        if 'train_english' in subdir:
            y= y[abs(y )>0.01]

            mat = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=64, n_fft =int(sr*0.025), hop_length = int(sr*0.010))
            preprocess_train_english = mat[:,0:1000].T
            train_seq_eng = np.append(train_seq_eng, preprocess_train_english)
        elif 'train_hindi' in subdir:
            y  = y [abs(y )>0.01]

            mat = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=64, n_fft =int(sr*0.025), hop_length = int(sr*0.010))
            preprocess_train_hindi = mat[:,0:1000].T
            train_seq_hindi = np.append(train_seq_hindi, preprocess_train_hindi)
        else:
            y = y[abs(y)>0.008]

            mat = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=64, n_fft =int(sr*0.025), hop_length = int(sr*0.010))
            preprocess_train_Mand = mat[:,0:1000].T
            train_seq_Mand = np.append(train_seq_Mand, preprocess_train_Mand)
```

7. English audio samples: 164 Number of Hindi audio samples: 41 Number of Mandarin audio samples: 110
8. Reshaping of data is done into (164,1000,64), (41,1000,64) and (110,1000,64) respectively for English, Hindi, and Mandarin.

```python
#Reshaping
English_SEQ = np.reshape(train_seq_eng, [164,1000,64])
Labels_eng = np.zeros([164,1000,1])
Hindi_SEQ = np.reshape(train_seq_hindi, [41,1000,64])
Labels_Hindi = np.ones([41,1000,1])
Mand_SEQ = np.reshape(train_seq_Mand, [110,1000,64])
Labels_Mand = np.ones([110,1000,1])*2

Train_data = np.vstack([English_SEQ,Hindi_SEQ, Mand_SEQ])
labels = np.vstack([Labels_eng, Labels_Hindi, Labels_Mand])
np.save('Train_data', Train_data)
np.save('Labels', labels)
```

9. Features and labels are concatenated. Now the tensor size will become (315,1000,64).
10. Samples are shuffled to ensure that validation split contain data from all classes.
11. Training is done.

```python
#Train Model
training_in_shape = Train_data.shape[1:]
training_in = Input(shape = training_in_shape)
Var = GRU(1240, return_sequences=True, stateful = False)(training_in)
training_pred = Dense(3, activation = 'softmax')(Var)
```

```python
[10] training_model = Model(inputs = training_in, outputs = training_pred)
     training_model.compile(loss = keras.losses.SparseCategoricalCrossentropy(),
                            optimizer = 'adam',
                            metrics = ['accuracy'])
     training_model.summary()
```

```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 1000, 64)]        0
_________________________________________________________________
gru (GRU)                    (None, 1000, 1240)        4858320
_________________________________________________________________
dense (Dense)                (None, 1000, 3)           3723
=================================================================
Total params: 4,862,043
Trainable params: 4,862,043
Non-trainable params: 0
_________________________________________________________________
```

12.
13. Streaming Model:

```python
##### define the streaming-infernece model
streaming_in = Input(batch_shape=(1,None,64))  ## stateful ==> needs batch_shape specified
foo = GRU(1240, return_sequences=False, stateful=True )(streaming_in)
streaming_pred = Dense(3, activation = 'softmax')(foo)
streaming_model = Model(inputs=streaming_in, outputs=streaming_pred)

streaming_model.compile(loss = keras.losses.SparseCategoricalCrossentropy(),
                      optimizer = 'adam',
                      metrics = ['accuracy'])
streaming_model.summary()


###### copy the weights from trained model to streaming-inference model
training_model.save_weights('weights.hdf5', overwrite=True)
streaming_model.load_weights('weights.hdf5')
keras.utils.plot_model(training_model, to_file='Streaming_Model.png', show_shapes=True, show_layer_names=True)
```

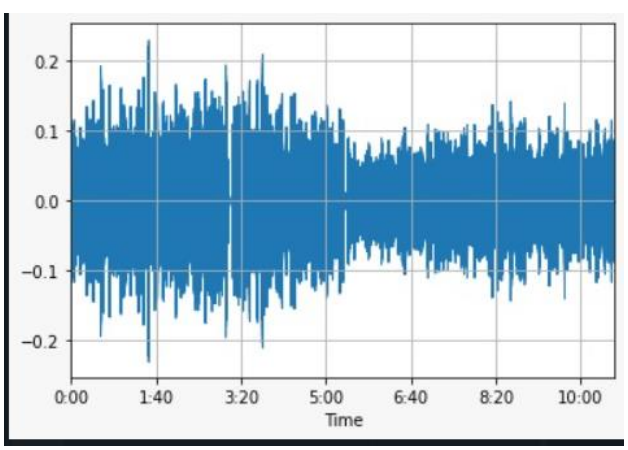
```
DEMO = 1
if DEMO:
    ##### demo the behaivor
    print('\n\n******the streaming-inference model can replicate the sequence-based trained model:\n')
    for s in range(1):
        print(f'\n\nRunning Sequence {s} with STATE RESET:\n')
        for n in range(20):
            in_feature_vector = Train_data[s][n].reshape(1,1,64)
            single_pred = streaming_model.predict(in_feature_vector)[0]
            print(single_pred)
            streaming_model.reset_states()

            p1 = plt.scatter(n, single_pred[0], color = 'blue')
            p2 = plt.scatter(n, single_pred[1], color = 'red')
            p3 = plt.scatter(n, single_pred[2], color = 'green')
    p1.set_label('English')
    p2.set_label('Hindi')
    p3.set_label('Mandarin')
    plt.savefig('model_plot.png')
    plt.legend()
    plt.show()
```

```
******the streaming-inference model can replicate the sequence-based trained model:
```

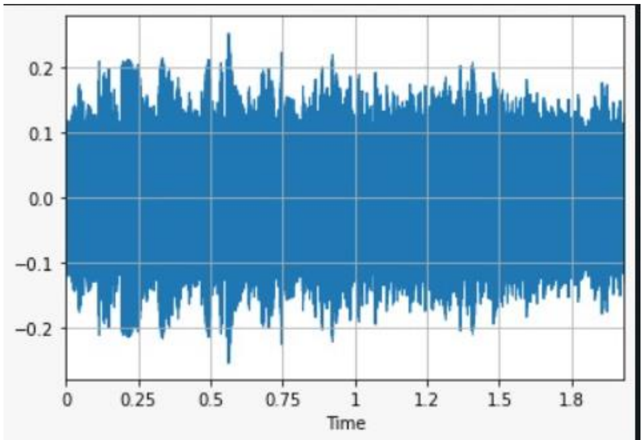
Thus, the class imbalance is dealt with assigning weights to the Cross-Entropy loss function during the training process. The weights are assigned such that the class having high number of samples gets the least weights. I have used recurrent neural network model with a single GRU layer comprising 1240 units and a three-neuron output layer with softmax activation. The labels used are integers: 0-English, 1-Hindi and 2- Mandarin, hence I have used sparse categorical cross entropy loss function to probability scores of languages at the output.

Handling silence:

Original Audio:



After handling silence:



I observed the amplitude wave-plot of the audio samples and ignored those whose amplitude fell below a threshold of 0.01 for English and Hindi and 0.008 in case of Mandarin. The differences in threshold is because mandarin audio samples had lower amplitudes on average.

ACCURACY:

Overall validation accuracy is 65%.

```
Epoch 2/20
63/63 [==============================] - 15s 234ms/step - loss: 0.8927 - accuracy: 0.5673 - val_loss: 0.8761 - val_accuracy: 0.5873
Epoch 3/20
63/63 [==============================] - 15s 236ms/step - loss: 0.8539 - accuracy: 0.6046 - val_loss: 0.8593 - val_accuracy: 0.5967
Epoch 4/20
63/63 [==============================] - 15s 238ms/step - loss: 0.8264 - accuracy: 0.6261 - val_loss: 0.8297 - val_accuracy: 0.6298
Epoch 5/20
63/63 [==============================] - 15s 239ms/step - loss: 0.7994 - accuracy: 0.6475 - val_loss: 0.8114 - val_accuracy: 0.6413
Epoch 6/20
63/63 [==============================] - 15s 238ms/step - loss: 0.7820 - accuracy: 0.6565 - val_loss: 0.8004 - val_accuracy: 0.6473
Epoch 7/20
63/63 [==============================] - 15s 236ms/step - loss: 0.7613 - accuracy: 0.6713 - val_loss: 0.7938 - val_accuracy: 0.6501
Epoch 8/20
63/63 [==============================] - 15s 236ms/step - loss: 0.7475 - accuracy: 0.6786 - val_loss: 0.7905 - val_accuracy: 0.6499
Epoch 9/20
63/63 [==============================] - 15s 236ms/step - loss: 0.7323 - accuracy: 0.6881 - val_loss: 0.8110 - val_accuracy: 0.6333
Epoch 10/20
63/63 [==============================] - 15s 237ms/step - loss: 0.7278 - accuracy: 0.6870 - val_loss: 0.7849 - val_accuracy: 0.6491
Epoch 11/20
63/63 [==============================] - 15s 238ms/step - loss: 0.7118 - accuracy: 0.6969 - val_loss: 0.7730 - val_accuracy: 0.6612
Epoch 12/20
63/63 [==============================] - 15s 237ms/step - loss: 0.6968 - accuracy: 0.7057 - val_loss: 0.7747 - val_accuracy: 0.6596
Epoch 13/20
63/63 [==============================] - 15s 237ms/step - loss: 0.6926 - accuracy: 0.7062 - val_loss: 0.7726 - val_accuracy: 0.6576
Epoch 14/20
63/63 [==============================] - 15s 237ms/step - loss: 0.6727 - accuracy: 0.7196 - val_loss: 0.7767 - val_accuracy: 0.6586
Epoch 15/20
63/63 [==============================] - 15s 237ms/step - loss: 0.6641 - accuracy: 0.7226 - val_loss: 0.7724 - val_accuracy: 0.6625
Epoch 16/20
63/63 [==============================] - 15s 237ms/step - loss: 0.6552 - accuracy: 0.7252 - val_loss: 0.7691 - val_accuracy: 0.6637
Epoch 17/20
63/63 [==============================] - 15s 237ms/step - loss: 0.6632 - accuracy: 0.7170 - val_loss: 0.7884 - val_accuracy: 0.6534
Epoch 18/20
63/63 [==============================] - 15s 237ms/step - loss: 0.6476 - accuracy: 0.7277 - val_loss: 0.7715 - val_accuracy: 0.6608
Epoch 19/20
63/63 [==============================] - 15s 237ms/step - loss: 0.6203 - accuracy: 0.7454 - val_loss: 0.7856 - val_accuracy: 0.6589
Epoch 20/20
63/63 [==============================] - 15s 237ms/step - loss: 0.6106 - accuracy: 0.7499 - val_loss: 0.7825 - val_accuracy: 0.6507
```

Result on test data:

```
        name = 'Mandarin'

    print('Prediction: {}, Max Index : {}, Prediction: {}'.format(single_pred, np
    streaming_model.reset_states()
```

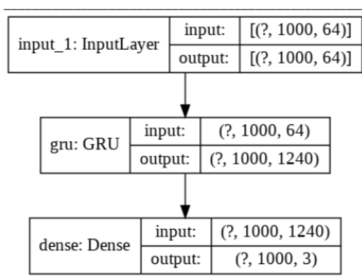
```
The streaming-inference model can replicate the sequence-based trained model:


Running Sequence 0 with STATE RESET:

English      Hindi       Mandarin

WARNING:tensorflow:Model was constructed with shape (None, 1000, 64) for input Tensor("in
Prediction: [[0.7583821  0.02539995 0.216218  ]], Max Index : 0, Prediction: English
Prediction: [[0.60263926 0.22253855 0.17482221]], Max Index : 0, Prediction: English
Prediction: [[0.63491344 0.07662525 0.28846127]], Max Index : 0, Prediction: English
Prediction: [[0.2665335  0.01279825 0.72066826]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.2728637  0.04354166 0.6835946 ]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.51125485 0.02896157 0.45978358]], Max Index : 0, Prediction: English
Prediction: [[0.30871066 0.05231581 0.63897353]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.36580116 0.00868511 0.62551373]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.2827969 0.0126734 0.7045297]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.7052199  0.01718601 0.2775941 ]], Max Index : 0, Prediction: English
Prediction: [[0.81110406 0.05188017 0.13701573]], Max Index : 0, Prediction: English
Prediction: [[0.6931776  0.08494528 0.22187707]], Max Index : 0, Prediction: English
Prediction: [[0.25416514 0.02826169 0.71757317]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.2114893  0.01936655 0.7691442 ]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.37646157 0.01707131 0.60646707]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.3901495  0.03890307 0.5709474 ]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.5072111  0.02462237 0.46816656]], Max Index : 0, Prediction: English
Prediction: [[0.5307878  0.07257678 0.39663538]], Max Index : 0, Prediction: English
Prediction: [[0.5559606  0.05412057 0.3899188 ]], Max Index : 0, Prediction: English
Prediction: [[0.19961709 0.00997883 0.79040414]], Max Index : 2, Prediction: Mandarin
```

Inference:

I observed that English language is classified more accurately than others. Mandarin is sometimes confused by English and accuracy on Hindi is very less, it largely confused with Mandarin.

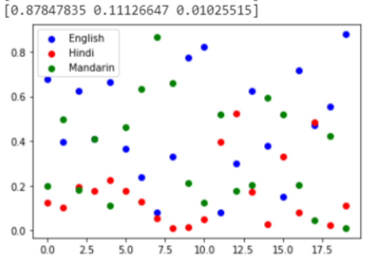
MODEL PLOT:

```
Model: "functional_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(1, None, 64)]           0
_________________________________________________________________
gru_1 (GRU)                  (1, 1240)                 4858320
_________________________________________________________________
dense_1 (Dense)              (1, 3)                    3723
=================================================================
Total params: 4,862,043
Trainable params: 4,862,043
Non-trainable params: 0
_________________________________________________________________
```



Three output 3 scores (probabilities) for each 10 msec input feature-vector – i.e., the probability of English, Hindi, and Mandarin

```
Running Sequence 0 with STATE RESET:

[0.67665035 0.12242746 0.20092219]
[0.39797    0.10419966 0.49783042]
[0.62348    0.19339027 0.18312977]
[0.41122323 0.17810647 0.41067025]
[0.6642383  0.2237378  0.11202388]
[0.36370924 0.17547193 0.46081886]
[0.23948018 0.12746783 0.63305205]
[0.0808401  0.0538922  0.86526775]
[0.33085865 0.01074776 0.6583936 ]
[0.774194   0.01497133 0.21083461]
[0.8234371  0.05134393 0.12521906]
[0.08177238 0.397115   0.5211126 ]
[0.30090886 0.5225376  0.17655356]
[0.623402   0.17184678 0.20475125]
[0.38063926 0.02606308 0.59329766]
[0.15136564 0.33030823 0.5183261 ]
[0.7159335  0.08181728 0.20224917]
[0.46971613 0.48360056 0.04668334]
[0.5566758  0.02232899 0.42099524]
[0.87847835 0.11126647 0.01025515]
```



```
import librosa
import os
import numpy as np
import pandas as pd
import glob, os
from dask.distributed import LocalCluster
from dask.diagnostics import ProgressBar

import argparse
import matplotlib.pyplot as plt
import shutil
from tensorflow.keras.layers import Input, Dense, GRU, Dropout
from tensorflow.keras import Model
import tensorflow
from tensorflow import keras

from tensorflow.keras import regularizers
from sklearn.utils import class_weight
from tensorflow.keras.models import load_model
```

```
# Mount GDrive

from google.colab import drive
drive.mount('/content/drive')
```

```
     Mounted at /content/drive
```

```
import os
os.chdir("/content/drive/My Drive/EE599_HW6")
```

```
top_db=30
train_file = '/content/drive/MyDrive/EE599_HW6/train/'
for subdir, dirs, files in os.walk(train_file):
    for file in files:
      #print(subdir)
        print(file)
        y, sr = librosa.load(subdir + "/" + file ,sr=16000)
        # print(y)
        # intervals = librosa.effects.split(y, top_db=top_db)
        # y_new = np.zeros((1))
        # for interval in intervals:
        #     y_new = np.concatenate((y_new, y[interval[0]: interval[1]]))
        # print(y_new)
        if 'train_english' in subdir:
            y= y[abs(y )>0.01]
```

```
        mat = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=64, n_fft =int(sr*0.025), hop_lengt
        preprocess_train_english = mat[:,0:1000].T
        train_seq_eng = np.append(train_seq_eng, preprocess_train_english)
    elif 'train_hindi' in subdir:
        y  = y [abs(y )>0.01]

        mat = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=64, n_fft =int(sr*0.025), hop_lengt
        preprocess_train_hindi = mat[:,0:1000].T
        train_seq_hindi = np.append(train_seq_hindi, preprocess_train_hindi)
    else:
        y = y[abs(y)>0.008]

        mat = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=64, n_fft =int(sr*0.025), hop_lengt
        preprocess_train_Mand = mat[:,0:1000].T
        train_seq_Mand = np.append(train_seq_Mand, preprocess_train_Mand)
```

```
english_0072.wav
english_0066.wav
english_0099.wav
english_0106.wav
english_0112.wav
english_0113.wav
english_0107.wav
english_0098.wav
english_0067.wav
english_0073.wav
english_0077.wav
english_0063.wav
english_0088.wav
english_0103.wav
english_0117.wav
english_0116.wav
english_0102.wav
english_0089.wav
english_0062.wav
english_0076.wav
english_0060.wav
english_0074.wav
english_0048.wav
english_0114.wav
english_0100.wav
english_0101.wav
english_0115.wav
english_0049.wav
english_0075.wav
english_0061.wav
english_0006.wav
english_0012.wav
english_0013.wav
english_0007.wav
english_0039.wav
```

```
english_0011.wav
english_0005.wav
english_0004.wav
english_0010.wav
english_0038.wav
english_0014.wav
english_0028.wav
english_0029.wav
english_0001.wav
english_0015.wav
english_0003.wav
english_0017.wav
english_0016.wav
english_0002.wav
english_0027.wav
english_0033.wav
english_0032.wav
english_0026.wav
english_0018.wav
english_0030.wav
english_0024.wav
english_0025.wav
english_0031.wav
english_0019.wav
```

```
print(train_seq_eng.shape)
print(train_seq_hindi.shape)
print(train_seq_Mand.shape)

print(preprocess_train_Mand.shape)
```

```
(7744000,)
(2112000,)
(5440000,)
(1000, 64)
```

```
from sklearn.model_selection import train_test_split
X_train, X_test = train_test_split(Train_data,test_size=0.2)
y_train, y_test = train_test_split(labels,test_size=0.2)
```

```
Train_data = np.load('Train_data.npy')
labels = np.load('Labels.npy')
```

```
print(labels.shape)
```

```
(315, 1000, 1)
```

```
labelsf = np.reshape(labels, [315*1000, 1])
dataf = np.reshape(Train_data, [315*1000, 64])
rng_state = np.random.get_state()
np.random.shuffle(dataf)
np.random.set_state(rng_state)
np.random.shuffle(labelsf)
labels = np.reshape(labelsf, [315, 1000, 1])
Train_data = np.reshape(dataf, [315, 1000, 64])



print(Train_data .shape)
print(labels.shape)
```

```
(315, 1000, 64)
(315, 1000, 1)
```

```
lab = np.reshape(labels, [315*1000,])
class_weights = class_weight.compute_class_weight('balanced', np.unique(lab), lab)


# ######  Define/Build/Train Training Model
# training_in_shape = x.shape[1:]
# training_in = Input(shape=training_in_shape)
# # training_in = Input(batch_shape=(None,train_seq_length,feature_dim)) this works too
# foo = GRU(4, return_sequences=True, stateful=False)(training_in)
# training_pred = Dense(1)(foo)

# training_model = Model(inputs=training_in, outputs=training_pred)
# training_model.compile(loss='mean_squared_error', optimizer='adam')
# training_model.summary()

# training_model.fit(x, y, batch_size=2, epochs=100)


#Train Model
training_in_shape = Train_data.shape[1:]
training_in = Input(shape = training_in_shape)
Var = GRU(1240, return_sequences=True, stateful = False)(training_in)
training_pred = Dense(3, activation = 'softmax')(Var)


training_model = Model(inputs = training_in, outputs = training_pred)
training_model.compile(loss = keras.losses.SparseCategoricalCrossentropy(),
                      optimizer = 'adam',
                      metrics = ['accuracy'])
training_model.summary()
```

```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 1000, 64)]        0
_________________________________________________________________
gru (GRU)                    (None, 1000, 1240)        4858320
_________________________________________________________________
dense (Dense)                (None, 1000, 3)           3723
=================================================================
Total params: 4,862,043
Trainable params: 4,862,043
Non-trainable params: 0
_________________________________________________________________
```

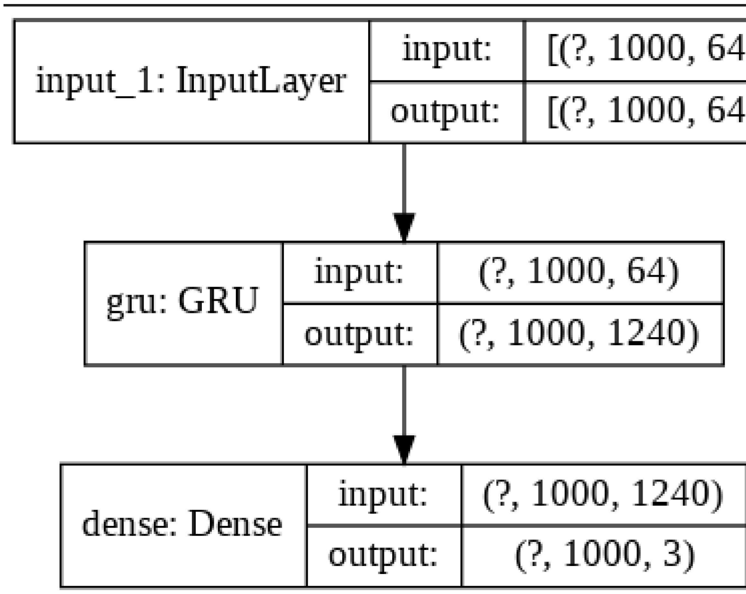
```
results = training_model.fit(Train_data, labels, batch_size = 4, epochs = 20,
                             validation_split = 0.2)
```

```
Epoch 1/20
63/63 [==============================] - 15s 238ms/step - loss: 1.1106 - accuracy: 0.49
Epoch 2/20
63/63 [==============================] - 15s 233ms/step - loss: 0.8969 - accuracy: 0.56
Epoch 3/20
63/63 [==============================] - 15s 233ms/step - loss: 0.8572 - accuracy: 0.60
Epoch 4/20
63/63 [==============================] - 15s 235ms/step - loss: 0.8250 - accuracy: 0.63
Epoch 5/20
63/63 [==============================] - 15s 237ms/step - loss: 0.8016 - accuracy: 0.64
Epoch 6/20
63/63 [==============================] - 15s 239ms/step - loss: 0.7822 - accuracy: 0.66
Epoch 7/20
63/63 [==============================] - 15s 241ms/step - loss: 0.7660 - accuracy: 0.66
Epoch 8/20
63/63 [==============================] - 15s 243ms/step - loss: 0.7491 - accuracy: 0.67
Epoch 9/20
63/63 [==============================] - 15s 242ms/step - loss: 0.7392 - accuracy: 0.68
Epoch 10/20
63/63 [==============================] - 15s 241ms/step - loss: 0.7257 - accuracy: 0.69
Epoch 11/20
63/63 [==============================] - 15s 241ms/step - loss: 0.7171 - accuracy: 0.69
Epoch 12/20
63/63 [==============================] - 15s 241ms/step - loss: 0.7052 - accuracy: 0.70
Epoch 13/20
63/63 [==============================] - 15s 242ms/step - loss: 0.6946 - accuracy: 0.70
Epoch 14/20
63/63 [==============================] - 15s 242ms/step - loss: 0.6839 - accuracy: 0.71
Epoch 15/20
63/63 [==============================] - 15s 242ms/step - loss: 0.6688 - accuracy: 0.72
Epoch 16/20
63/63 [==============================] - 15s 242ms/step - loss: 0.6626 - accuracy: 0.72
Epoch 17/20
63/63 [==============================] - 15s 241ms/step - loss: 0.6486 - accuracy: 0.73
Epoch 18/20
63/63 [==============================] - 15s 241ms/step - loss: 0.6410 - accuracy: 0.73
Epoch 19/20
```

```
63/63 [==============================] - 15s 242ms/step - loss: 0.6443 - accuracy: 0.72
Epoch 20/20
63/63 [==============================] - 15s 242ms/step - loss: 0.6323 - accuracy: 0.73
```

```
##### define the streaming-infernece model
streaming_in = Input(batch_shape=(1,None,64))  ## stateful ==> needs batch_shape specified
foo = GRU(1240, return_sequences=False, stateful=True )(streaming_in)
streaming_pred = Dense(3, activation = 'softmax')(foo)
streaming_model = Model(inputs=streaming_in, outputs=streaming_pred)

streaming_model.compile(loss = keras.losses.SparseCategoricalCrossentropy(),
                      optimizer = 'adam',
                      metrics = ['accuracy'])
streaming_model.summary()


###### copy the weights from trained model to streaming-inference model
training_model.save_weights('weights.hdf5', overwrite=True)
streaming_model.load_weights('weights.hdf5')
keras.utils.plot_model(training_model, to_file='Streaming_Model.png', show_shapes=True, show_
```

```
Model: "functional_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(1, None, 64)]           0
_________________________________________________________________
gru_1 (GRU)                  (1, 1240)                 4858320
_________________________________________________________________
dense_1 (Dense)              (1, 3)                    3723
=================================================================
Total params: 4,862,043
Trainable params: 4,862,043
Non-trainable params: 0
_________________________________________________________________
```



```
DEMO = 1
if DEMO:
    ##### demo the behaivor
    print('\n\n******the streaming-inference model can replicate the sequence-based trained m
    for s in range(1):
        print(f'\n\nRunning Sequence {s} with STATE RESET:\n')
        for n in range(20):
            in_feature_vector = Train_data[s][n].reshape(1,1,64)
            single_pred = streaming_model.predict(in_feature_vector)[0]
            print(single_pred)
            streaming_model.reset_states()

            p1 = plt.scatter(n, single_pred[0], color = 'blue')
            p2 = plt.scatter(n, single_pred[1], color = 'red')
            p3 = plt.scatter(n, single_pred[2], color = 'green')
    p1.set_label('English')
    p2.set_label('Hindi')
    p3.set_label('Mandarin')
    plt.savefig('model_plot.png')
    plt.legend()
    plt.show()
```

```
******the streaming-inference model can replicate the sequence-based trained model:



Running Sequence 0 with STATE RESET:

[0.5302123  0.04146109 0.42832664]
[0.37607756 0.03238284 0.59153956]
[0.37506357 0.01354169 0.61139476]
[0.64063865 0.09303318 0.26632822]
```

```
training_model.save('EE599_HW5_training_charu.hdf5')
```

```
[0.1775959  0.00889719 0.81350696]
```

```
training_model.save('EE599_HW5_Streaming_charu.hdf5')
```

```
[0.697043   0.2392142  0.06374282]


[0.6346321  0.18265682 0.18271104]
```

```
#Provide Val_data with dimensions (Num_sequences, Num_samples, 64)
Num_samples=10
Num_sequences=1
import librosa
import os
import numpy as np
import pandas as pd
import glob, os
from dask.distributed import LocalCluster
from dask.diagnostics import ProgressBar

import argparse
import matplotlib.pyplot as plt
import shutil
from tensorflow.keras.layers import Input, Dense, GRU, Dropout
from tensorflow.keras import Model
import tensorflow
from tensorflow import keras

from tensorflow.keras import regularizers
from sklearn.utils import class_weight
from tensorflow.keras.models import load_model



streaming_model = load_model('EE599_HW5_Streaming_charu.hdf5')

DEMO = 1
if DEMO:
    ##### demo the behaivor
    print('\n\n******the streaming-inference model can replicate the sequence-based trained m
    for s in range(Num_sequences):
```

```
    print(f'\n\nRunning Sequence {s} with STATE RESET:\n')
    for n in range(Num_samples):
        in_feature_vector = X_test[s][n].reshape(1,1,64)
        single_pred = streaming_model.predict(in_feature_vector)[0]
        print(single_pred)
        streaming_model.reset_states()
```

```
******the streaming-inference model can replicate the sequence-based trained model:



Running Sequence 0 with STATE RESET:

WARNING:tensorflow:Model was constructed with shape (None, 1000, 64) for input Tensor("
[[0.38893133 0.23447867 0.37659   ]]
[[0.5037544  0.06291688 0.43332875]]
[[0.3884066  0.5120579  0.09953551]]
[[0.23267122 0.02893938 0.73838943]]
[[0.33690926 0.14555919 0.5175315 ]]
[[0.16905035 0.04825482 0.78269476]]
[[0.7032599  0.09518144 0.2015587 ]]
[[0.2191856  0.13904268 0.6417717 ]]
[[0.6701619  0.24929874 0.08053936]]
[[0.11615907 0.07504063 0.8088003 ]]
```

```
import librosa
import os
import numpy as np
import glob
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Input, Dense, GRU, Dropout
from tensorflow.keras import Model
import tensorflow
from tensorflow import keras

from tensorflow.keras.models import load_model


test_english = './train/train_english/english_0001.wav'
test_hindi = './train/train_hindi/hindi_0004.wav'
test_mandarin = './train/train_mandarin/mandarin_0001.wav'

y, fs = librosa.load(test_hindi , sr = 16000)
y = y[abs(y)>0.01]
sr = 16000
mat = librosa.feature.mfcc(y=y,sr = sr, n_mfcc=64, n_fft =int(sr*0.025), hop_length = int(sr*
preprocess = mat[:,0:1000].T
# print(preprocess.shape)

streaming_model = load_model('EE599_HW5_Streaming_charu.hdf5')
```

```
Num_sequences = 1
Num_samples =  20
DEMO = 1
if DEMO:
    ##### demo the behaivor
    print('\n\n The streaming-inference model can replicate the sequence-based trained model:
    for s in range(Num_sequences):
        print(f'\n\nRunning Sequence {s} with STATE RESET:\n')
        print('English      Hindi       Mandarin\n')
        for n in range(Num_samples):
            in_feature_vector = preprocess[n].reshape(1,-1,64)
            single_pred = streaming_model.predict(in_feature_vector)[0]
            pred_class = np.argmax(single_pred)
            if pred_class == 0:
              name = 'English'
            elif pred_class == 1:
              name = 'Hindi'
            else:
              name = 'Mandarin'

            print('Prediction: {}, Max Index : {}, Prediction: {}'.format(single_pred, np.arg
            streaming_model.reset_states()
```

```
 The streaming-inference model can replicate the sequence-based trained model:



Running Sequence 0 with STATE RESET:

English      Hindi       Mandarin

WARNING:tensorflow:Model was constructed with shape (None, 1000, 64) for input Tensor("
Prediction: [[0.15329592 0.02002116 0.8266829 ]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.35048252 0.03352675 0.6159907 ]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.13426997 0.0339576  0.83177245]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.32979155 0.07054628 0.5996622 ]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.3518918  0.04038556 0.60772264]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.04708453 0.00448222 0.9484333 ]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.32183522 0.01890745 0.65925735]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.32500857 0.01960263 0.6553888 ]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.25671294 0.01061882 0.7326682 ]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.4428394  0.07646763 0.4806929 ]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.87026274 0.0808876  0.04884961]], Max Index : 0, Prediction: English
Prediction: [[0.79194254 0.03221401 0.17584348]], Max Index : 0, Prediction: English
Prediction: [[0.7004332  0.12780975 0.17175706]], Max Index : 0, Prediction: English
Prediction: [[0.58403516 0.11343447 0.30253038]], Max Index : 0, Prediction: English
Prediction: [[0.08903258 0.04699596 0.8639714 ]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.02594464 0.03008636 0.943969  ]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.11339858 0.10230232 0.7842991 ]], Max Index : 2, Prediction: Mandarin
Prediction: [[0.5025427  0.03496958 0.46248776]], Max Index : 0, Prediction: English
Prediction: [[0.6502214  0.01880278 0.3309758 ]], Max Index : 0, Prediction: English
Prediction: [[0.5103056  0.03612767 0.45356676]], Max Index : 0, Prediction: English
```